Creating Custom ASP.NET Controls in VS.NET

1.1.1.1. Contents

2. Overview....2
2.1. Goals 2
2.2. Scenarios 2
3. Creating a Custom Web Control in VS.NET....3
3.1. New project 3
3.2. Add Items templates 3
3. Implementing Design View Support....6
3.1. Rendering in Design View 6
3.2. Implementing a Custom Control Designer 6
3.2.1. Specifying a Designer for a Control....7
3.2.2. Creating a Custom Control Designer....7
3.2.3. Customizing the Design View Representation of the Control....7
3.3. Implementing Toolbox support 9
3.3.1. Showing the control in toolbox....9
3.3.2. Specifying the default tag XML....9
3.3.3. Specifying the TagPrefix....9
3.3.4. Specifying the toolbox icon....10
3.4. Implementing Property Grid support 11
3.5. Implementing IntelliSense support 12
3.6. Packaging and deploying the control 12
4. Using a Custom Web Control in VS.NET....13
4.1. Installing the control on the Control User's machine 13
4.2. Adding the control to the toolbox 13
4.3. Using the control on a Web Form 13
4.3.1. Adding a control manually in HTML....13
4.4. Deploying the control to the App Domain 13

2. Overview

This document describes the technical users can employ to author and consume custom Web controls for ASP.NET Web applications using Visual Studio.NET.

Creating custom ASP.NET server controls is much like creating custom Windows Forms controls. Support for the shared functionality (custom value editors, designer verbs, etc.) is described in the Programming Guide section of the .NET Frameworks documentation. This document describes the design-time support that is unique to ASP.NET server control authoring.

More information about authoring custom ASP.NET controls can be found in the .NET Framework SDK documentation set: **Building .NET Frameworks Applications > ASP.NET Web Forms > Developing ASP.NET Server Controls**.

2.1. Goals

- Allow third-party control authors to build and package custom controls for use within a Web Forms application. The packaging includes compiled code to help control authors protect their intellectual property.
- Allow Visual Studio.NET users to easily add third-party controls to the toolbox, and use them in a Web Forms application.
- Help the user deploy the control and its dependent files to the web server when the application is run from the IDE.

2.2. Scenarios

2.2.1.1. ISV develops a custom Web control

Bob works for BobWare, a software component vendor. He creates a custom ASP.NET server control called “BobTable.” Bob uses the information found in the .NET Frameworks documentation (Programming Guide) to create and package up his control in a form he can distribute to other developers. The control package includes the compiled code for the control, image resources, and some string resources. For the rest of this document, we call Bob a *Control Vendor*.

2.2.1.2. Corporate developer develops a custom Web control for use in her own application

Susan works for MultiMegaCorp as a developer for the company’s Web applications. She creates a custom ASP.NET server control called “MegaWidget.” She starts VS.NET and uses the **Customize Toolbox** dialog to install the MegaWidget control, then uses it like other controls in the creation of a Web application. Susan deploys her control only as a part of the application she is creating: it never gets packaged. For the rest of this document, we call Susan a *Control Author*.

2.2.1.3. Another developer uses the control in a Web Forms application

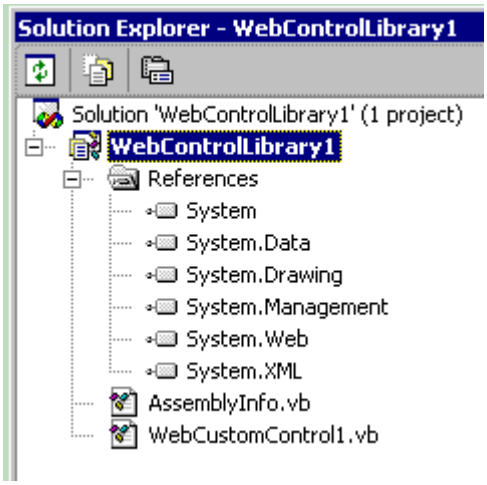
Helen works for IPOSoonOrDie.com as a web application developer. Because her application will use tables in innovative ways, she purchases a copy of BobTable. She installs the package from BobWare, then starts VS.NET and uses the **Customize Toolbox** dialog to add the BobTable control. The BobTable icon appears on her toolbox. She drops instances of the control on her Web Forms to build her application. In the rest of this document, Helen is the *Control User*.

3. Creating a Custom Web Control in VS.NET

Note: This section describes the VS.NET project support for creating a custom Web control. Other tools may implement this support differently.

3.1. New project

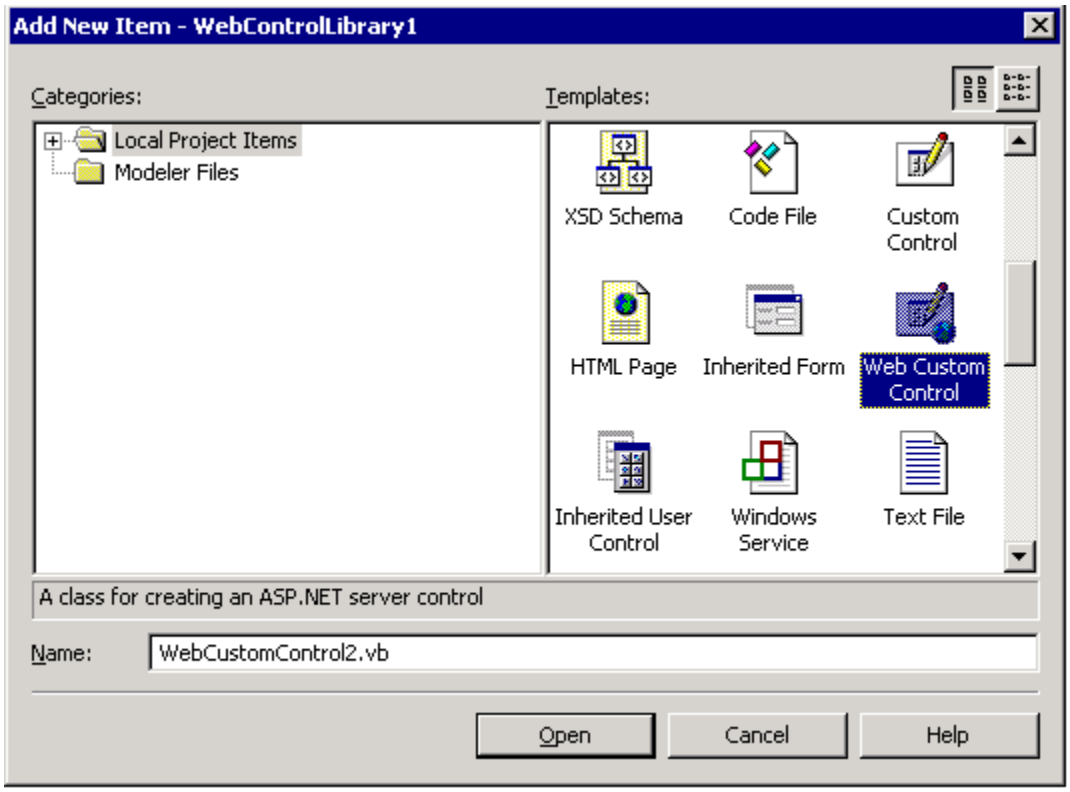
Visual Studio.NET supports creating custom ASP.NET server controls in either VB or C#. To create a new custom Web control, first select the language for the project. Then select the **Web Control Library** project type template. A new application is created with the following files and references:



The **WebCustomControl** item in the project includes a stub control as a working example. Note that this is a client project; no Web application is created for this project.

3.2. Add Items templates

A custom Web control is created from the **Web Custom Control** template. This template is available in VB and C# client projects.



The Web Custom Control template has the following content:

VB.NET:

```
Imports System.ComponentModel
Imports System.Web.UI

<DefaultProperty("Text"), ToolboxData("<{0}:WebCustomControl1 runat=server></{0}:WebCustomControl1>")> _
Public Class WebCustomControl1
    Inherits System.Web.UI.WebControls.WebControl

    Dim _text As String

    <Bindable(True), Category("Appearance"), DefaultValue("")> Property [Text]() As String
        Get
            Return _text
        End Get

        Set(ByVal Value As String)
            _text = Value
        End Set
    End Property

    Protected Overrides Sub Render(ByVal output As System.Web.UI.HtmlTextWriter)
        output.Write([Text])
    End Sub

End Class
```

C#.NET:

```
using System;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.ComponentModel;

namespace WebControlLibrary2
{
    [DefaultProperty("Text"),
        ToolboxData("<{0}:WebCustomControl1 runat=server></{0}:WebCustomControl1>")]
    public class WebCustomControl1 : System.Web.UI.WebControls.WebControl
    {
        private string text;

        [Bindable(true),
            Category("Appearance"),
            DefaultValue("")]
        public string Text
        {
            get
            {
                return text;
            }

            set
            {
                text = value;
            }
        }

        protected override void Render(HtmlTextWriter output)
        {
            output.Write(Text);
        }
    }
}
```

3. Implementing Design View Support

Visual Studio.NET has a WSIWYG design surface for ASP.NET page development called “Design View.” This section describes implementing support for this designer surface, the property grid and the toolbox. VS.NET is the specific example used, but the designer support a control author implements is actually generic. Third parties tools that implement .NET design time support can also surface the designer support build into a control and it’s designer.

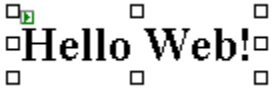
3.3. Rendering in Design View

In order to render in Design View in Visual Studio.NET, a custom Web control uses a designer that derives from the System.Web.UI.Design.ControlDesigner class. This class provides default designer support for all ASP.NET server controls (including those that derive directly from System.Web.UI.Control).

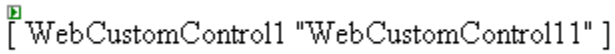
The default ControlDesigner simply calls the control’s Render() method to generate the HTML that defines the appearance of the control in Design view. For example, this simple control:

```
using System;
using System.Web.UI;
using System.ComponentModel;

namespace SimplestControl {

   public class ReallySimple : System.Web.UI.WebControls.WebControl {

      protected override void Render(HtmlTextWriter output) {
         output.Write("<H2>Hello Web!</H2>");
      }
   }
}
```

Renders this in VS.NET Design view:



Sometimes, a control renders only the values of one or more of its properties (this is the case with the stub control in the WebCustomControl template). So, if no value is set, the HTML string that defines control’s appearance is empty. In this case, the default ControlDesigner calls it’s GetEmptyDesignTimeHtml() method, which generates a representation composed of the type and ID of the control:



3.4. Implementing a Custom Control Designer

A custom control designer is an additional, independent class that derives from the System.Web.UI.Design.ControlDesigner class. It is used at design time only, permits the control writer to:

- display a different representation of the control at design time than at runtime
- add actions to the context menu for the control; for example launch a custom property page. See .NET Framework help topic “Designer Verbs”.

3.4.1. Specifying a Designer for a Control

A control specifies it's designer via the **DesignerAttribute** on the control's runtime class. For example:

```
using System;
using System.Web.UI;
using System.ComponentModel;

namespace SimpleControl {

   [
      Designer("SimpleControl.Design.SimpleDesigner, SimpleControl")
   ]
   public class Simple : System.Web.UI.WebControls.WebControl {

      private string text;

      public string Text {

         get { return text; }
         set { text = value; }
      }

      protected override void Render(HtmlTextWriter output) {
         output.Write(Text);
      }
   }
}
```

For more information about the DesignerAttribute class, see the .NET Frameworks Reference documentation.

3.4.2. Creating a Custom Control Designer

A custom control designer may derive from one of the following classes:

System.Web.UI.Design.**ControlDesigner**	General purpose designer for controls that derive from Control and WebControl
System.Web.UI.Design.WebControls.**TemplatedControlDesigner**	Adds designer support for template editing
System.Web.UI.Design.WebControls.**ReadWriteControlDesigner**	Adds designer support for in-place editing (like asp:Panel)

By convention, the control designer resides in a namespace that extends that of the control. For example, if the control resides in the "SimpleControl" namespace, it's designer should reside in the "SimpleControl" or "SimpleControl.Design" namespace. It can exist in the same assembly or a different assembly. Creating a single assembly for both the control and designer is considerably easier to distribute and install, however the size of the designer code in the control's DLL will increase the overall size of your application. If it's large, you may wish to consider splitting the designer into a separate assembly.

Note: The presence of the designer code in the runtime DLL has no impact on runtime performance or working set.

These base classes are implemented in System.Design.DLL, so you'll need to add a compilation reference to this assembly for your project. In VS.NET, use Project > Add Reference.

3.4.3. Customizing the Design View Representation of the Control

You can customize the HTML that represents the control at design time by overriding the GetDesignTimeHtml() method, or only when the HTML is blank by overriding the GetEmptyDesignTimeHtml() method. These methods return an HTML string to the designer that is used to represent the control.

One way to create the HTML is to temporarily override the properties on this control, or set properties on another control, then call it's RenderControl() method. The rendered output is collected by an HtmlTextWriter, and returned as the string result of the GetDesignTimeHtml() method. For example:

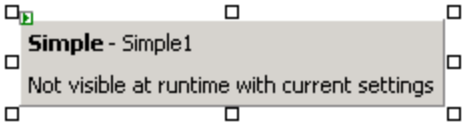
```
using System;
using System.IO;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.Design;

namespace Simple.Design {

   public class SimpleDesigner : ControlDesigner {

      public SimpleDesigner() {
      }

      public override string GetDesignTimeHtml() {

         // Component is the control instance, defined in the base designer
         Simple simple = (Simple) Component;

         StringWriter sw = new StringWriter();
         HtmlTextWriter tw = new HtmlTextWriter(sw);

         HyperLink placeholderLink = new HyperLink();

         // put control text into the link's Text
         placeholderLink.Text = simple.Text;
         placeholderLink.RenderControl(tw);

         return sw.ToString();
      }
   }
}
```

Here's an example of a simple designer that displays it's runtime HTML when present, but provides an alternate representation when the Render method does not produce any HTML at design time. This example uses the CreatePlaceHolderDesignTimeHtml() method (defined in the base ControlDesigner) to create the "grey block" style placeholder used for User Controls, DataList and Repeater.

```
using System;
using System.Web;
using System.Web.UI;
using System.Web.UI.Design;

namespace Simple.Design {

   public class SimpleDesigner : ControlDesigner {

      public SimpleDesigner() {
      }
```

```
    protected override string GetEmptyDesignTimeHtml() {

        // when blank, render design time warning
        string text="Not visible at runtime with current settings";
        return CreatePlaceHolderDesignTimeHtml(text);
    }
  }
}
```

This designer renders the following when the design-time HTML is empty:



3.5. Implementing Toolbox support

3.5.1. Showing the control in toolbox

Due to changes in Beta2, all Web controls are displayed correctly when added to the toolbox. It's no longer necessary to implement the **ShowInToolbox** or **ToolboxItemAttribute** attributes – the necessary support is already implemented in the base Control class.

3.5.2. Specifying the default tag XML

The attribute **ToolboxData** is used to define the tag XML that will be injected into the page when an end-user drags and drops the control to a Web Form. In Beta2, it's no longer necessary to implement this attribute, the default drop data is automatically generated.

However, you can implement this attribute to override the default data. This example drops not only the tag name, but also a property setting:

```
using System;
using System.Web.UI;
using System.ComponentModel;

namespace SimpleControl {

  [
     ToolboxData("<{0}:Simple Text=”Hello World” runat=server></{0}:Simple>")
  ]
  public class Simple : System.Web.UI.WebControls.WebControl {

    private string text;

    public string Text {

      get { return text; }
      set { text = value; }
    }

    protected override void Render(HtmlTextWriter output) {
      output.Write(Text);
    }
  }
}
```

The tag prefix for the control is substituted for the {0} token at design time. The ID attribute is also automatically added to the tag at design time.

3.5.3. Specifying the TagPrefix

In Beta2 and beyond, dragging a control from the toolbox automatically injects a <%@ Register %> directive for the control into the page, if one of the control's namespace doesn't yet exist. You can set the preferred TagPrefix for your namespace in your assembly. This will be used to create the register directive for your namespace, unless the TagPrefix is already used in the page by a different namespace. (In that case, a numeric digit will be appended to the TagPrefix to "uniquify" it.)

If the TagPrefix attribute is not specified, VS.NET will generate a tag prefix beginning with the letters "cc": cc1, cc2, etc.

The **TagPrefixAttribute** is and assembly-level attribute. If your assembly includes controls in more than one namespace, you should add a TagPrefix attribute for each namespace.

```
using System;
using System.Web.UI;
using System.ComponentModel;

[ assembly:TagPrefix("SimpleControl", "simple") ]

namespace SimpleControl {

   [
      Designer("SimpleControl.Design.SimpleDesigner, SimpleControl")
   ]
   public class Simple : System.Web.UI.WebControls.WebControl {

      private string text;

      public string Text {

         get { return text; }
         set { text = value; }
      }

      protected override void Render(HtmlTextWriter output) {
         output.Write(Text);
      }
   }
}
```

Note: VS.NET supplies an AssemblyInfo class for setting assembly-level attributes. You can set TagPrefix there as well.

For more information about the TagPrefixAttribute class, see the .NET Frameworks reference documentation.

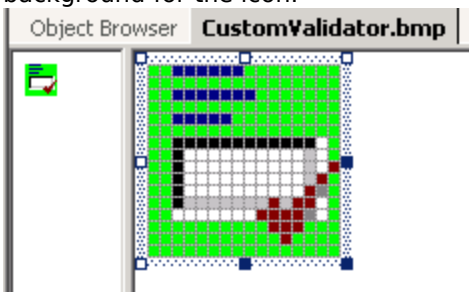
3.5.4. Specifying the toolbox icon

The VS.NET toolbox will look in the control's assembly for an embedded image to display as the toolbox icon. The image must satisfy all of these requirements:

- Must be a bitmap (.BMP) file
- Format must be 16x16 pixels, 16-bit color
- Filename must match the control's class name. For example: Simple.bmp.

When creating the bitmap, you will probably wish the background of the image to be transparent so your icon's background is the color of the toolbox regardless of user desktop settings. VS.NET uses the pixel in the lower left corner to determine the color that should become transparent when the

icon is displayed in the designer. In this example, the green color will become a transparent background for the icon:



If you are creating the control in VS.NET, it's easy to mark the image as an embedded resource. Add the BMP file to the same project as the control, and set it's name to match the controls class. Then select the BMP file in Solution Explorer, and set it's **Build Action** property in the Properties window to "Embedded Resource".

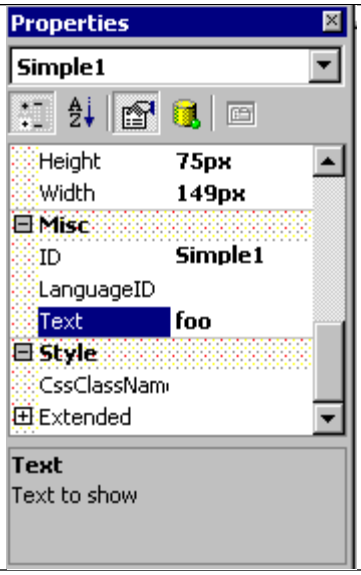
3.6. Implementing Property Grid support

Attributes are also applied on a per-property basis to define how the property will be exposed via the property grid, and how changes made in the property grid will be persisted. For example:

```
using System;
using System.Web;
using System.Web.UI;

namespace SimpleControl {

   public class Simple2 : WebControl {
     private String _text;

     [
         Bindable(true),
         Category("Misc"),
         DefaultValue(""),
         Description("Text to show"),
     ]
     public string Text {
         get {
            return _text;
         }
         set {
            _text = value;
         }
     }

     protected override void Render(HtmlTextWriter output) {
        output.Write("<H2>" + _text + "</H2>");
     }
   }
}
```

Several attributes are used to define the behavior of the property grid:

Bindable	Whether a property should be display in the [DataBindings...] dialog
Category	If the property grid is sorted by category, this determines which category the property should appear in.
DefaultValue	The default value of the property.
Description	Text describing the property. This appears at the bottom of the property gird in the “Description” box.
PersistenceMode	How (or whether) changes made to the value of this property should be persisted.
Browsable	Whether a property is displayed in the designer
TypeConverter, Editor	Hooks up extended UI for setting the property



Please consult the reference documentation for the .NET Frameworks for more details about these attributes.

3.7. Implementing IntelliSense support

Visual Studio’s HTML view can provide statement completion for third-party controls, provided they install an XML schema that defines the valid elements and attributes for the control’s XML tag. The schema must implement specific elements -- see the asp.xsd file installed by VS.NET for an example.

3.8. Packaging and deploying the control

The *control vendor’s* installation package must deliver the control runtime and optional designer components to the developer’s machine. The Windows Logo program requires ISVs to install software to a subfolder of the target machine’s Program Files folder. For example:

C:\\Program Files\Common Files\BobWare\1.0

The control’s assembly can be referenced by path from this directly. If the control’s designer is compiled into the same assembly as the control, no additional registration steps are required. However, if a designer is installed in a separate assembly, an “AssemblyFolder” key must be added to the registry in order for VS.NET to find it. For example:

[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\.NET\AssemblyFolders\BobWare]
@=" C:\\Program Files\\Common Files\\BobWare\\1.0\\”

Control vendors should insert this key in the HKLM node; *control authors* may register where VS.NET can locate their component lists under HKCU.

A *control author* may choose to skip the formal packaging and registration steps above, particularly when the control will be used only on his own machine. The simplest packaging is to put both the control and it’s designer in a single assembly. That way, VS.NET setup consists only of adding a the control to the toolbox.

Alternatively, a control author can put a copy of the designer’s assembly in the VS.NET Common7\ IDE directory. This location holds many of the component assemblies for VS.NET.

Note: We do not recommend that control vendors install designers to this folder; to do so is a violation of Windows Logo requirements.

4. Using a Custom Web Control in VS.NET

4.1. Installing the control on the Control User's machine

The *Control User* must install the control and its dependent files to some location on her development machine. If the control has a custom designer, and the designer is not included in the same assembly as the control, the designer's assembly must be registered by the installation package, as described in the previous section.

4.2. Adding the control to the toolbox

When a control package is installed the control does not automatically become visible in the VB7 IDE. The user uses the **Customize Toolbox** dialog to install the packaged control into the VB7 IDE. Installing the control into the IDE makes it visible on a tab of the toolbox and therefore available for use on a Web Form. The toolbox retains the path to the control's assembly.

4.3. Using the control on a Web Form

In VS.NET, when the user can drag the custom control from the toolbox to a Web Page, several actions automatically occur:

- The control's TagPrefix and Namespace are registered in the page.
- A tag is generated for the control, and added to the page.
- A reference to the control's assembly is added to the project. This has the effect of causing the control's assembly to be deployed to the \bin of the application.
- If applicable, the control's designer is automatically loaded.
- The control is displayed in design view.

4.3.1. Adding a control manually in HTML

The user can also manually add the control to a Web Form by typing it's tag in HTML view. In this case, the Register directive and a reference to the control's assembly are not automatically added to the project.

4.4. Deploying the control to the App Domain

Once the control's assembly is referenced by the VS.NET project, no further action is required to deploy the control with the project to the Web application's App Domain. This will happen as a part of the normal build/deploy process.